Generics

Programação Orientada a Objeto II

O que são Generics?

O conceito de Generics surgiu no Java 5 para diminuir a quantidade de bugs causados em cenários onde era necessário fazer cast constantemente.

Generics permite que tipos (classes e interfaces) sejam parametrizados na assinatura de métodos, classes e interfaces.

Isso possibilita a reutilização de código com segurança de tipo, evitando a necessidade de conversões explícitas e reduzindo erros de tempo de execução.

Benefícios dos Generics

- **Segurança de tipo:** Evita erros de *ClassCastException* em tempo de execução.
- **Reutilização de código:** Permite criar classes e métodos flexíveis.
- **Legibilidade:** Facilita a compreensão do código, tornando-o mais expressivo.

Problemática

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Perceba que a lista *list* armazena itens do tipo Object.

Portanto, sempre que um item é consumido precisa fazer o *casting* para o tipo original.

Porém, são permitidas aberrações do tipo:

```
List list = new ArrayList();
list.add("hello");

String s = (String) list.get(0);
list.add(1);

Integer num = (Integer) list.get(1);
```

Esse tipo de código torna ainda mais passível de bugs relacionados à nossa variável `list`.

Visto que dois tipos diferentes estão sendo misturados na mesma coleção.

Solução

Com a criação dos *Generics* e a possibilidade de parametrizar os tipos, agora escrevemos de acordo com a sintaxe Type<Parameter>. Ou seja, aquele primeiro trecho de código seria re-escrito como:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);
```

Tipos Genericos

Generics

Como você pôde perceber, a interface List é um tipo genérico que permite a parametrização.

```
//Assinatura da interface java.util.List

/**
Type parameters:
<E>- the type of elements in this list
*/
public interface List<E> extends Collection<E>
```

Tipos Genericos

Generics

Considere uma classe não-genérica Box que possui um atributo de nome "object".

```
public class Box {
   private Object object;

   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```

Tipos Genericos

Generics

Considere uma classe não-genérica Box que possui um atributo de nome "object".

```java
/**
* Generic version of the Box class.
* @param <T> the type of the value being boxed
*/
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Tipos Genericos

Generics

Considere uma classe não-genérica Box que possui um atributo de nome "object".

```
Box<Integer> integerBox = new Box<>();
```

Convenções

Generics

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

E - Element (usado exclusivamente pelo Java Collections Framework)

- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Convenções

Generics

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- E - Element (usado exclusivamente pelo Java Collections Framework), representa um elemento individual

```
class ListaElementos<E> {
    private E elemento;

    public void setElemento(E elemento) {
        this.elemento = elemento;
    }

    public E getElemento() {
        return elemento;
    }
}
```

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- K (Key) - Representa uma chave em estruturas de dados como Map<K, V>.

```
class Par<K, V> {
    private K chave;
    private V valor;

    public Par(K chave, V valor) {
        this.chave = chave;
        this.valor = valor;
    }

    public K getChave() { return chave; }
    public V getValor() { return valor; }
}
```

Convenções

Generics

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- N (Number) - Indica que o tipo é um número.

```
class Calculadora<N extends Number> {
    private N numero;

    public Calculadora(N numero) {
        this.numero = numero;
    }

    public double getDobro() {
        return numero.doubleValue() * 2;
    }
}
```

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- T (Type) - Representa um tipo genérico arbitrário.

```
class Caixa<T> {
    private T objeto;

    public void setObjeto(T objeto) {
        this.objeto = objeto;
    }

    public T getObjeto() {
        return objeto;
    }
}
```

Convenções

Generics

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- V (Value) - Representa um valor associado a uma chave (Map<K, V>).

```
class Armazem<V> {
    private V valor;

    public void setValor(V valor) {
        this.valor = valor;
    }

    public V getValor() {
        return valor;
    }
}
```

Convenções

Generics

Em Java, algumas convenções são amplamente utilizadas ao nomear tipos genéricos:

- S, U, V, etc. - Usados para representar múltiplos tipos genéricos em uma classe ou método.

```
class Conversor<S, U> {
    private S entrada;
    private U saida;

    public Conversor(S entrada, U saida) {
        this.entrada = entrada;
        this.saida = saida;
    }

    public S getEntrada() { return entrada; }
    public U getSaida() { return saida; }
}
```

Métodos genéricos

Generics

Assim como definimos tipos genéricos, também podemos definir métodos genéricos.

Sabemos que para comparar dois elementos do tipo inteiro usamos os operadores de comparação `>` ou `<`. Porém, eles não são válidos para comparar textos, datas ou outros objetos. Para esses casos, existe a interface `Comparable`.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Métodos genéricos

Generics

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Veja que, antes declaravámos na assinatura de um método:

- o modificador de visibilidade,
- o tipo de retorno,
- o nome do método e
- seus argumentos

Agora, temos um novo elemento de Generics `<T extends Comparable<T>>`.

Com isso, definimos que o método é parametrizável. Mas, não só isso. Também estamos definindo que o tipo `T` em questão deve, obrigatoriamente, implementar a `interface Comparable`.

1. Métodos estáticos não podem acessar parâmetros de tipo da classe:

Se uma classe é genérica *(class MinhaClasse<T>)*, seus métodos estáticos não podem usar T. No entanto, métodos estáticos podem ter seus próprios parâmetros genéricos:

```
class Util {
   public static <T> void imprimir(T elemento)
{
        System.out.println(elemento);
   }
}
```

2. Não é possível instanciar diretamente tipos genéricos:

```
class Caixa<T> {
    // Isso NÃO é permitido:
    // private T objeto = new T();
}
```

Em vez disso, o objeto precisa ser passado ou criado de outra forma.

3. Não é possível usar tipos primitivos diretamente com Generics:

```
Caixa<int> caixaDeInteiros;   // ERRO!
```

Você deve usar wrappers (Integer, Double, etc.):

```
Caixa<Integer> caixaDeInteiros;
```

Exercícios

Exercício 1: Carrinho de compras e pagamento

1. **Criação de um carrinho de compras genérico:**

- Implemente uma classe Carrinho<T> que permita armazenar itens genéricos.
- Deve conter métodos para adicionar e remover itens do carrinho.
- Deve haver um método que retorne a quantidade total de itens.

Dica: Utilize uma **List<T>** para armazenar os itens do carrinho.

Exercício 1: Carrinho de compras e pagamento

2. Sistema de pagamento genérico:

- Crie uma interface Pagamento<T> com um método processarPagamento(T valor).
- Implemente duas classes PagamentoCartao e PagamentoPix que aceitem valores diferentes.
- Simule uma transação processando um pagamento com cada método.

Dica: Utilize **T extends Number** para garantir que o valor seja numérico.

Exercício 1: Carrinho de compras e pagamento

3. Gestão de pedidos genérica:

- Implemente uma classe Pedido<K, V> que armazene um identificador do pedido e uma lista de itens.
- O identificador deve ser genérico (K), podendo ser um número ou código alfanumérico.
- A lista de itens deve ser um conjunto de objetos (V).

Dica: Utilize um **Map<K, List<V>>** para armazenar os pedidos.

Exercício 2: Classe genérica para diferentes tipos de cupons de desconto

Cenário: O e-commerce precisa de uma funcionalidade para aplicar cupons de desconto a diferentes categorias de produtos.

Requisitos:

- Crie uma classe CupomDesconto<T extends Number>, onde T pode ser um Integer (para descontos fixos) ou Double (para percentuais).
- Adicione um método aplicarDesconto(T desconto, double precoOriginal) que retorne o preço final após o desconto.

Obrigada